KubeCon Observability Day 2026

Hallucinating High Availability

Where AI-driven remediation breaks down



Before we begin

The demo and code shown today are my own work.

They do not represent my employer’s technical stack.

The promise: faster triage, fewer late-night pages

A highly confident suggestion can still be catastrophically wrong.

The next 20 minutes

The replay

A rolling update that looks healthy, and the fix an assistant suggests

Where it breaks

The same agent, with different documentation and different tools

The blueprint

Evidence validation, deterministic guardrails and rollback criteria

One bad release, three services

`payment-gateway`

1000 ms

New v2 client timeout

→

`user-auth`

1500 ms

Its normal response time

→

`auth-db`

Healthy

Separate Postgres deployment

How we made it behave like production

Real database
SQL through a 5-connection pool, deployed apart from releases

Real cancellation
When the gateway hangs up, the healthy service aborts too

GitOps only
Argo CD self-heals, so every lasting change is a pull request

Shared release train
Two services in one commit, and one carries a security fix

Tail-sampled traces
30 s until a trace appears, with all errors and 10% of healthy kept

SLO alerts
Burn-rate rules in Prometheus, routed through Alertmanager

The cluster says healthy. Users say otherwise.

Pods	Running, Ready
Restarts	0
Readiness	Passing
Argo CD	Synced, Healthy

100%

of payment requests return 503

The probe checks a health endpoint that never calls the dependency.

Every log line is true. None says who's at fault.

```
payment-gateway
ERROR calling user-auth: Client.Timeout exceeded while awaiting headers

user-auth
WARN auth: request aborted after 1100ms during verify-token-db: context canceled
```

The gateway gave up. Cancellation propagates, so the healthy dependency also looks like it failed.

A typical assistant blames the victim

> **“the user-auth v2 rollout added about 1.2s of database connection pool wait”**

```
kubectl rollout undo deployment/user-auth
```

4 of 5 runs with no documentation

It changes the wrong service

The gateway still gives up after one second.

It won’t stick

Argo CD self-heals and reverts direct changes.

It can remove a security fix

In today’s app, that release carries one.

Same gateway timeout. Only the documentation and tools changed.

Documentation	Tools added	Steps	Root cause the agent gave	Evidence that decided it	Result	Fix it proposed	Tokens
No documentation 5 runs, earlier app version	None	11–17	4 of 5 blamed `user-auth`: "the user-auth v2 rollout added about 1.2s of database connection pool wait"	Pod logs and deployment state. Tool output was not saved.	✗ 1 of 5 correct	4 of 5 used direct `kubectl`, which Argo CD reverts	not logged
General KB 5 runs, earlier app version	None	9–13	5 of 5 blamed `user-auth` v2 waiting on the database pool	Pod logs and deployment state. Tool output was not saved.	✗ 0 of 5 correct	5 of 5 used Git, as the KB describes	not logged
General KB 1 run	Prometheus	27	"(hypothesis, not confirmed): the unreleased auth-db migration is holding a lock on tokens"	**Logs** showed a lock left by an earlier test. **Metrics** showing only v2 failing went unused.	✗ Wrong	Inspect the database, terminate the migration	642,374
KB with false claims 1 run	Prometheus	25	"The payment-gateway v2 image gives up on user-auth after about 1.1 s"	**Logs** showed the client timeout. **Metric history** of gateway v1 with `user-auth` v2 cleared `user-auth`, a window left by an earlier rollback test.	△ Correct, inferred	Roll back the gateway only	464,180
General KB 1 run	Prometheus Tempo	17	"user-auth:v2 … adds about 1.5s of latency" from the `pg_sleep` in its query	**Rollout timing** only. Its own logs showed the same 1.5 s query before v2. One trace search, no span comparison.	✗ Wrong	Fix or roll back `user-auth`	411,280
Blueprint* 1 run	Prometheus Tempo	9	"payment-gateway:v2 has a compiled-in client timeout of 1000 ms", shorter than the 1500 ms `user-auth` needs	**Full trace:** gateway timed out at 1000 ms, `user-auth` canceled 104 ms later. **Metric history** gave the 1.5 s baseline.	✓ Correct, confirmed	Roll back the gateway only, through the gate	164,559

***Blueprint:** a service-specific runbook for this failure. It lists the normal latency, the steps for comparing gateway and `user-auth` trace spans, how to tell each failure case apart, and the approved rollback. It also states the 1000 ms timeout value.

What the Blueprint told the agent

The runbook the agent read with `fetch_skill`, excerpted word for word

payment-gateway-blueprint.md

Meta

tools: kubectl, prometheus, tempo, argocd

caution: NEVER recommend imperative kubectl scale, kubectl patch, or kubectl rollout undo … ArgoCD will immediately overwrite them.

1. Deterministic Topology & Baseline Contract

Expected Baseline Latency: a healthy /auth call takes about 1500ms … This is its steady state, not saturation.

Cancellation propagates: … A canceled user-auth span is a consequence of the caller leaving, and on its own it says nothing about user-auth health.

Release mechanics: payment-gateway:v1 = 2500ms, payment-gateway:v2 = 1000ms, overridable via CLIENT_TIMEOUT_MS …

Guardrail Rule: Do NOT apply historical KB scaling runbooks … unless Case C below is established.

Instructions in the same file

2. Mandatory OpenTelemetry Evidence Validation (Grafana Tempo)

Before recommending any remediation …

1. Query Grafana Tempo … for recent failed traces

2. Inspect the parent-child spans for the same trace_id, and **establish which side ended the call first:**

> **Case A (Frontend Timeout Regression):** the payment-gateway client span … ends with STATUS_ERROR and context deadline exceeded at about 1000ms, the child user-auth server span ends about 100 ms later with context canceled

> **Case B (Database Lock):** the payment-gateway client span ends with context deadline exceeded at about 2500ms … db_pool_wait rising

> **Case C (True user-auth Bottleneck):** the user-auth server span exceeds 2000ms or returns HTTP 5xx … Only then investigate user-auth saturation.

3. Remediation Criteria (ArgoCD + Prometheus SLI)

1. Verify the error rate breach in Prometheus.

2. **Case A only:** … recommend a surgical rollback of that service alone through the gate

3. **Never git revert the release train commit.**

> **Prompt, same for the four newest runs:** “Alerts are firing in namespace default. First read every firing alert … The team’s documentation for this service is available as the skill "payment-gateway". … Investigate all firing alerts together and determine the root cause, then state the exact remediation command(s).”

Highlighted: the Blueprint states the timeout value, so it shows the agent can follow a procedure, not work out the cause alone.

Inside the agent's trace: same tools, different path

Model call · Tool call · Documentation read · Tempo query

General KB, Prometheus, Tempo

55 s, 8 model calls, 17 tool calls. ✗ Wrong



Blueprint, Prometheus, Tempo

22 s, 4 model calls, 9 tool calls. ✓ Correct, confirmed



Blueprint run: read the procedure at 2 s, searched failed gateway traces at 5 s, opened one at 11 s.

General KB run: read logs and Kubernetes state first. Tempo was its last call at 38 s, on `user-auth` only, and it never opened a trace.

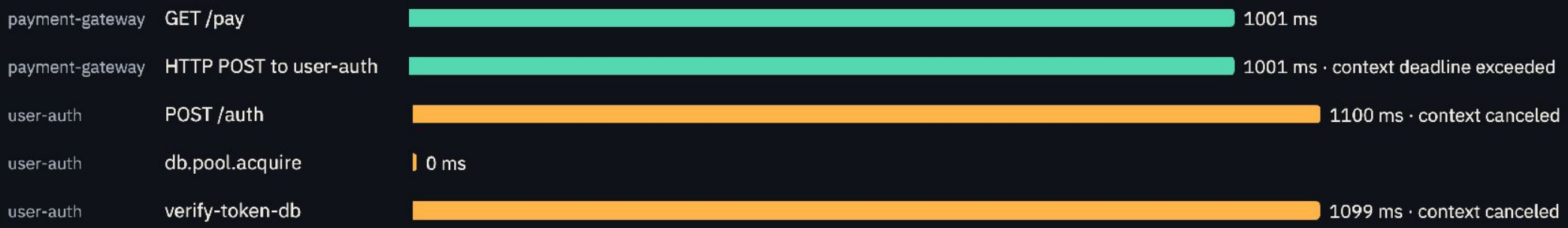
One run each, from the agent's OpenTelemetry spans in Tempo. All bars share one time axis.

What the agent traces show

Documentation and tools	Duration	Model calls	Tool calls	First Tempo	Failed or blocked	Last prompt	What the trace reveals
General KB Prometheus	115 s	11	**27** Logs 12 · Metrics 6 · Shell 6 · K8s 2 · Docs 1	Not enabled	3: blocked `psql`, pod not found, no data	90,711	12 log reads chased a lock left by an earlier test. Metrics split by version came back at +23.8 s and went unused.
KB with false claims Prometheus	91 s	9	**25** Logs 9 · Metrics 8 · K8s 7 · Docs 1	Not enabled	1: no data	82,700	False claims entered at +1.9 s. A 90-minute metric query at +36.2 s held the window that cleared `user-auth`.
General KB Prometheus, Tempo	55 s	8	**17** Metrics 5 · Logs 5 · K8s 3 · Shell 2 · Docs 1 · Traces 1	+38.0 s	0	73,677	Tempo was the last tool call. It searched `user-auth` only, and all 5 traces returned predated v2.
Blueprint Prometheus, Tempo	22 s	4	**9** Metrics 5 · Traces 2 · Docs 1 · K8s 1	+5.4 s	0	52,314	Searched failed gateway traces at +5.4 s, then opened one. Read no pod logs.

The request trace that settled it

Opened by the Blueprint run. The gateway gave up at 1000 ms, and `user-auth` was cancelled about 100 ms later with no pool wait.



Single runs, not rates. Time is seconds into the investigation. Last prompt is input tokens on the final model call. Model calls took 78% to 94% of each run. HealthChecks emit no spans.

What Prometheus added

Documentation and tools	Calls	What it asked Prometheus	What came back	How it was used	Result
General KB Prometheus	6	Alerts, request rate by version, 1-hour request rate, 1-hour lock waits	Only v2 returned 503s while v1 served 200s. Lock waits from an earlier test, ending at 15:59.	**Lock history** supported its wrong theory. **The version split** went unused.	✗ Wrong
KB with false claims Prometheus	8	Alerts, request rate by version, 90-minute request rate, lock waits	Gateway v1 with `user-auth` v2 was healthy from about 14:44 to 15:05. Current lock wait was 0.	**Decided the answer.** That window cleared `user-auth`, but it came from an earlier rollback test.	△ Correct, inferred
General KB Prometheus, Tempo	5	Alerts, request rate by version, request rate history, lock waits	Only about 5 minutes of history. Both v2 services failed from the same moment.	**Timing only.** It showed when failures began, and could not separate the two v2 services.	✗ Wrong
Blueprint Prometheus, Tempo	5	Alerts, error ratio, `user-auth` **latency history**, error history, pool usage	`user-auth` averaged about 1.50 s before the rollout. Errors went from 0 to 100% by 16:36.	**Confirmed the baseline.** 1.5 s was normal for `user-auth`, so it was not the cause.	✓ Correct, confirmed

Metrics show when and which release

Request rates by version tied the failure to v2, but both services shipped as v2.

Only the Blueprint asked for a baseline

No other run queried `user-auth` latency, so none could show that 1.5 s was normal.

Every run saw all firing alerts

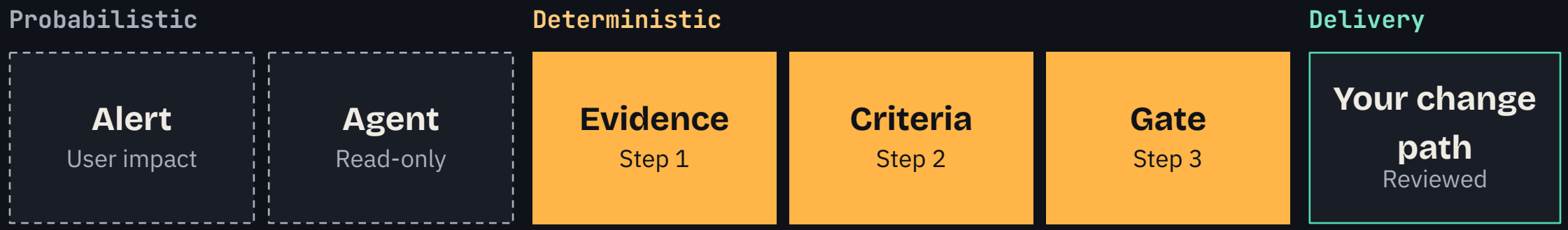
Each started with the same `ALERTS` query. No run used exemplars to reach a trace.

Single runs, not rates. The last two runs had about 5 minutes of metric history, after a Prometheus restart.

What we learned

In this demo	What we took from it
Pods stayed Ready while 100% of requests failed	**Cluster health doesn’t measure user impact**
One trace: the gateway timed out at 1000 ms, and user-auth was cancelled 104 ms later	**The order of failure settles blame**
Only one run looked up the normal 1.5 s latency	**A baseline only helps if something asks for it**
Failed traces took 30 s to appear	**An empty result is unknown, not healthy**
A 3-line rollback fixed the gateway and kept the security fix	**Change the smallest unit at fault**
Two gate runs gave 1 PR and a comment. A log-injection probe changed nothing.	**Deterministic checks hold when models don’t**

The agent proposes. Deterministic checks decide.



The model writes a **hypothesis**. Nothing it writes is executed.

Step 1

Validate the evidence

Who failed first?	Establish the order of failure across every service boundary.
Is this abnormal?	Compare with that component’s own history, not an assumption.
What changed?	Line up the onset with deployments, config, schema and infrastructure.
What would disprove it?	Each hypothesis names the observation that rules it out, then checks.

No matching evidence means no action. Missing data is “unknown”, not “healthy”.

Step 2

Act only when all four are true

Users are affected	Your SLO is breached by your own agreed threshold
The cause is proven	The evidence meets that failure’s written definition
The action fits the cause	It reverses the change that lines up with the onset
The blast radius is bounded	Use the smallest reversible unit, never a bundle of changes

Afterward, confirm the signal recovers. If it doesn’t, stop and escalate.

Step 3

Put a deterministic gate in the path

```
refuse unless:
action is on the allowlist
target exists in known state
change touches only expected fields
if already proposed: add evidence, no duplicate
```

Read-only agents

Investigation needs no write access.

Typed inputs only

The gate never parses model prose.

Your normal change path

Same review and audit as a person.

Next steps ...

01	**Keep agents read-only**	Let them propose, and never execute their text.
02	**Write the procedure**	For your top failures, name the evidence that separates them.
03	**Audit your docs**	Outdated or missing facts decide what the agent concludes.
04	**Trace the agent**	And redact those traces, because they contain your logs.
05	**Replay real incidents**	Score the agent against the known cause before you trust it.

Let the agent be wrong safely.

More telemetry didn’t give a better answer. A procedure changed which telemetry it used. Deterministic checks keep the wrong answers out of production.

Examples from small numbers of runs, not rates.